# Towards an Environment for doing Data Science that runs in Browsers

Leila Abidi, Christophe Cérin, Gilles Fedak, Haiwu He

# Towards an Environment for doing Data Science that runs in Browsers

Leila Abidi[1], Christophe Cérin[1], Gilles Fedak[2] and Haiwu He[3]

[1]Université de Paris 13, LIPN UMR CNRS 7030, 99, avenue Jean-Baptiste Clément, 93430 Villetaneuse, France
[2] Laboratoire de l'Informatique du Parallélisme, ENS Lyon, 46 avenue d'Italie, 69364 LYON CEDEX 07, France
[3] China Sciences & Technology Network, Computer Network Information Center, Chinese Academy of Sciences, 4 Zhongguancun Nansijie, Haidian District, Beijing 100190, ChinaP.O. Box 349
{leila.abidi,christophe.cerin}@lipn.univ-paris13.fr, gilles.fedak@inria.fr, haiwuhe@cstnet.cn

*Abstract*—**This article proposes a path for doing Data Science using browsers as computing and data nodes. This novel idea is motivated by the cross-fertilized fields of desktop grid computing, data management in grids and clouds, Web technologies such as Nosql tools, models of interactions and programming models in grids, cloud and Web technologies. We propose a methodology for the modeling, analyzing, implemention and simulation of a prototype able to run a MapReduce job in browsers. This work allows to better understand how to envision the big picture of Data Science in the context of the Javascript language for programming the middleware, the interactions between components and browsers as the operating system. We explain what types of applications may be impacted by this novel approach and, from a general point of view how a formal modeling of the interactions serves as a general guidelines for the implementation. Formal modeling in our methodology is a necessary condition but it is not sufficient. We also make round-trips between the modeling and the Javascript or used tools to enrich the interaction model that is the key point, or to put more details into the implementation. It is the first time to the best of our knowledge that Data Science is operating in the context of browsers that exchange codes and data for solving computational and data intensive programs. Computational and data intensive terms should be understand according to the context of applications that we think to be suitable for our system.**

*Index Terms*—**System design using formal modeling, Desktop grids, Data management, Web ecosystem.**

## I. Introduction

The increasing number of intelligent devices such as smartphones, tablets, watches and sensors in a broad sense, are new business opportunities but they also put a pressure to make existing applications to support these new devices and to remain competitive. Ten years ago, who could predict that Google Drive will going to supplement supplant desktop office suites? New Web and cloud technologies now provide feasible means to push almost any desktop functionality "on the Internet" and in most cases to run them in the user's browser. For instance, OfficeJS[1] is an open-source project similar to Google Drive to offer office suite. The development is with Javascript and it runs using the sesources of your Web browser. OfficeJS is partly using J-IO[2], a Javascript library that can be used to synchronize text files and that takes into account the offline mode.

Will it be possible soon to do Data Science in browsers? We mean here, to do computing and data management inside the user's browser, not in using the browser as a presentation interface. We do believe that an effort should be done earlier in the design stage for the interactions between the components of the system to be build to get confidence in the Internet-centric system. This paper proposes a methodology, illustrated by a concrete prototype for going into the direction of a full Data Science environment that runs in browsers, reaching in this way some form of universality. The main assumptions made in this paper are the following.

First, the browser is the 'universal' operating system and Javascript is the programming language for that operating system. This option is realistic regarding the number, types and of systems that are currently developed by the Web communities. Many efforts has been done for instance to secure the browser with some confinement mecanisms such as the ones presented in [1]. As a consequence, the browsers can run codes in a mode at least as secure than the mode using virtual machines.

Second, there is a class of applications that can benefit from the use of browsers for doing Data Science. In our case, we are working with medical doctors under the framework of the IDV Life Imaging framework[3]. One use case consider crowdsourcing for the annotation of images coming from scanners in the radioly field. Young doctors are invited to annotate images under the supervision of an expert. We could imagine to offer to the medical doctors the possibility to annote, but also to process images on the client side, i.e., in the browsers. In our case, the images are not very large in size. Another use case that we are working on is in the context of another project on context aware combuting, and as follows. Most of the mobile users now have multiple devices which further increases the opportunity to share their resources. However, these devices can be used only if they meet certain context requirements which come from requirements of the app that is running or from limits on the resource usage of the volunteered device. Imagine that you are watching a live

[1]http://www.erp5.com/officejs/documentation
[2]http://www.j-io.org/
[3]See: http://idv.parisdescartes.fr

match in the stadium. The allocated seat in the stadium only allows us one angle view. In case of specic event (a goal if you are watching a football game), you would like to view a $360^o$ video that is composed from the video feed from video cameras in all the four angles of the stadium.
Third, beyong the technology we need first a high level of abstraction, at least for reasoning about the system we are building, but also to guide the develoment. In this paper we focus on a very high level of abstraction for the modeling of the components and actors of the system we envision. We use Petri Nets for that purpose. We do not use them to prove some requirements of our systems because of the page limit but we have accomplished such work in [2] for instance. We illustrate how to get a smooth transition from the Petri Net to a more functional view of the system, then to a concrete prototype.

The first contribution is a methodology that considers the problem specification, analysis, implementation and simulation of the core components that the community consider as important for computing and managing data on the Internet, making real the opportunity for Data Science in the browser. The second contribution is a prototype demonstrating that the abstraction can be implemented in JavaScript in order to make the system universal (nowadays every devices are able to run JavaScript). We also isolate the limitations of the current JavaScript technologies in order to fully execute the System only inside the browser. Throughout this paper we would like to demonstrate that we are in search of a high level description for the interactions between components required for doing Data Science, that is somehow universal because it has been implemented yet for traditionnal computing platforms (grids, clusters, clouds) as well as on mobile computing platforms soon. Note that the work [2] is concurrently running for demonstrating the usefulness of our scheme for grids and clouds.

The organization of the paper is as follows. In section II we explain our abstraction for the main components of our system. We also provide with a discussion about the functional view and, according to this view we explain an execution of an application and how the components interact. In section III we give technical details about the prototype implementation. We also give some examples of the current limitations in the technology that lead to run some code outside the browser. The different classes that compose the system are documented. Section IV is about related works and it mainly focusses on grid data management because of the lack of researches, to the best of our knowledge, on using browsers for Data Science. Section V concludes the paper and draw some perspectives

## II. RedisDG as the abstraction for interactions

The Publish-Subscribe paradigm is an asynchronous mode for communicating between entities [3], [4]. Some users, namely the subscribers or clients or consumers, express and record their interests under the form of subscriptions, and are notified later by another event produced by other users, namely the producers.
This communication mode is multipoint, anonymous and implicit. Thus, it allows spatial decoupling (the interacting entities do not know each other), and time decoupling (the interacting entities do not need to participate at the same time). This total decoupling between the production and the consumption of services increases the scalability by eliminating many sorts of explicit dependencies between participating entities. Eliminating dependencies reduces the coordination needs and consequently the synchronizations between entities. These advantages make the communicating infrastructure well suited to the management of distributed systems and simplify the development of a middleware for the coordination of components in a Data Science context.
We also use Redis[4], hence the name RedisDG which is a no-SQL advanced key-value store with Publish-Subscribe functionality.

### A. *RedisDG modeling*

Our work started by performing a formal modeling, based on our initial modeling of the publish-subscribe paradigm [2] but adapted to Redis interactions for the Pub-Sub subsystem. In this former work we have studied an interaction scheme based on Bonjour from Apple which is a little bit different in the approach for publishing and subscribing to events.
The figure 1 illustrates the colored Petri net of the RedisDG system. We distinguish, in the center, the publish-subscribe paradigm based on the three transitions `Publish, Subscribe, Notify`. With Redis, if an event is published and there are no components that have previously subscribed to be notified by this event, then the event is lost forever. To achieve a faithful model of this behavior, we have adapted our basic publication-subscription model, to which we added control places in order to block a publication that has no subscription. The two control places are: `SubscribeControl` and `ModRedis`. Their roles consist to generate, update and compare the lists of published events and subscribed events. When implementing RedisDG, it was necessary to take into consideration this Redis specificity. The idea behind adding the control places was to always have the subscribers awaiting to be notified. This idea is a guideline for the development of RedisDG. Therefore, the potential workers in RedisDG, as soon as they join the system, must subscribe to an event. When we switch from the modeling to the code, a worker in RedisDG consists concretely into two *Threads*: the first manages the execution of tasks and the second manages the subscriptions to ensure the functioning of the system.

### B. *RedisDG protocol*

In this subsection, we introduce the coordination algorithm of RedisDG system. It is the highest view possible. Some technical details are given in the experiments section. The algorithm is entirely based on the publication-subscription paradigm. The middleware obtained offers the same features as the majority of desktop grid middleware such as Condor and BOINC. It manages scheduling strategies especially the dependencies between tasks, the execution of tasks and the verification/certification of the results; since the results returned

[4]See http://redis.io

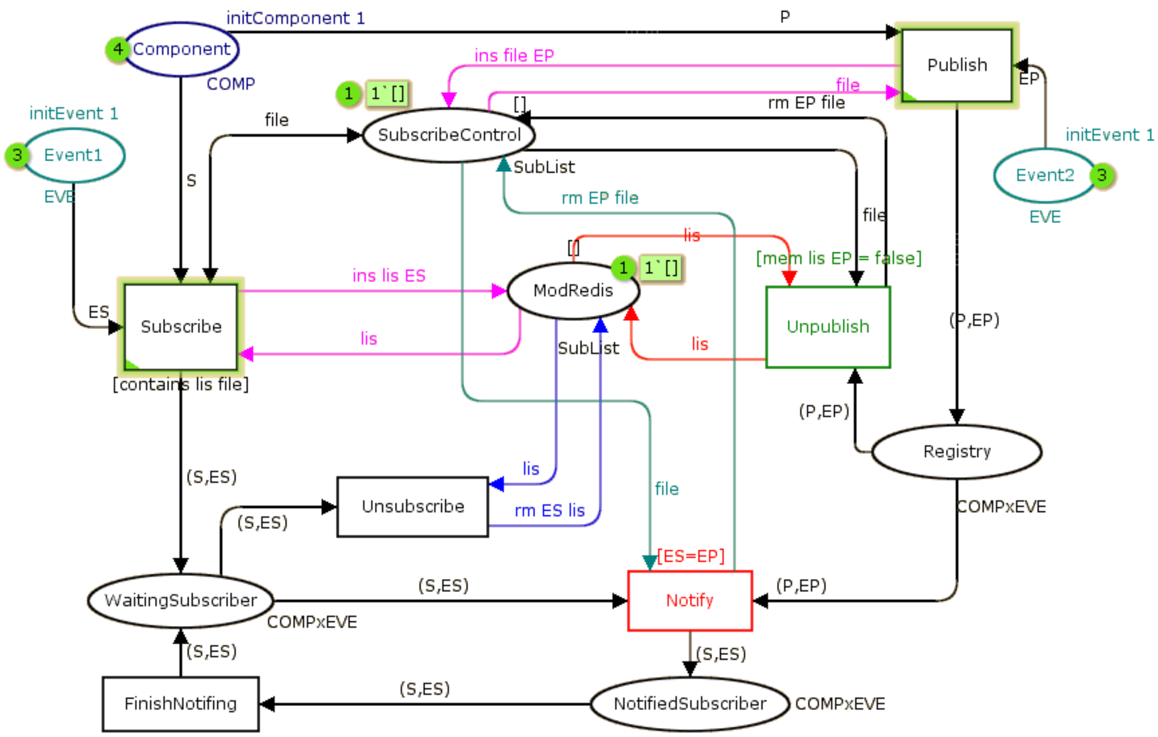


Fig. 1. Formal medeling of RedisDG

by the workers can be manipulated or altered by malicious workers. The general objectives for the RedisDG protocol are:

- Using an asynchronous paradigm (publish-subscribe) that ensures, as much as possible, a complete decoupling between the coordination steps (for performance reasons);
- Ensuring the system resilience by duplicating tasks and actors. Even if the system is asynchronous and the tasks are duplicated, we need to ensure the progress of tasks execution. We also assume that actors are duplicated for resilience reasons;

In Figure 2, we present the steps of an application execution. In RedisDG, a task may have five states: *WaitingTasks, TasksToDo, TasksInProgress, TasksToCheck* and *FinishedTasks*. These states are managed by five actors: a broker, a coordinator, a worker, a monitor and a checker. Taken separately, the behavior of each component in the system may appear simple, but we are rather interested in the coordination of these components, which makes the problem more difficult to solve.

The key idea is to allow the connection of dedicated components (coordinator, checker, ...) in a general coordination mechanism in order to avoid building a monolithic system. The behavior of our system as shown in Figure 2 is as follows:

1) Tasks batches submission. Each batch is a series-parallel graph of tasks to execute.
2) The Broker retrieves tasks and publishes them on the channel called *WaitingTasks*.
3) The Coordinator is listening on the channel *WaitingTasks*.
4) The Coordinator begins publishing independent tasks on the channel *TasksToDo*.
5) Workers announce their volunteering on the channel *VolunteerWorkers*.
6) The coordinator selects Workers according to SLA criteria.
7) The Workers, listening beforehand on the channel *TasksToDo* start executing the published tasks. The event 'execution in progress' is published on the channel *TasksInProgress*.
8) During the execution, each task is under the supervision of the Monitor whose role is to ensure the correct execution by checking if the node is alive. Otherwise the Monitor publishes again, tasks that do not arrive at the end of their execution. It puplishes, on the channel *TasksToDo*, in order to make the execution of the task done by other Workers.
9) Once the execution is completed, the Worker publishes the task on channel *TasksToCheck*.
10) The Checker verifies the result returned and publishes the corresponding task on the channel *FinishedTasks*.
11) The Coordinator checks dependencies between completed tasks and those waiting, and restarts the process in step (4).
12) Once the application is completed (no more tasks),

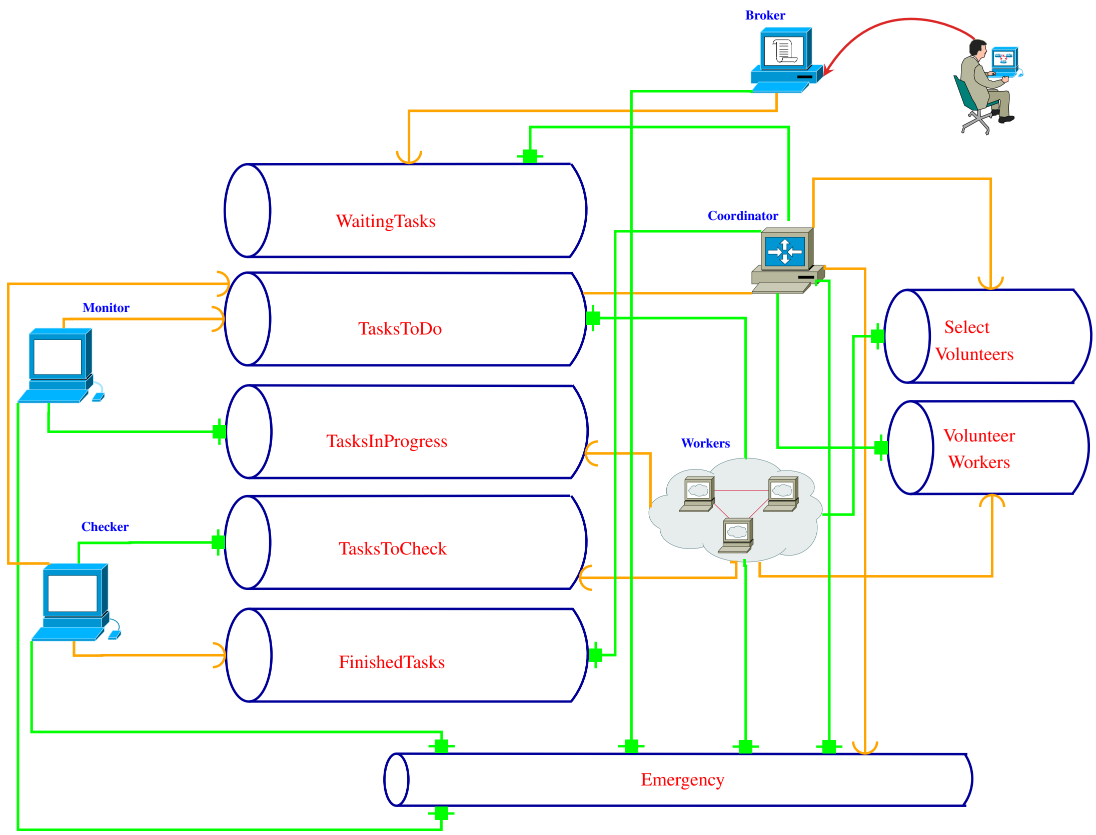


Fig. 2. Interactions between components of the RedisDG system

the Coordinator publishes a message on the channel *Emergency* to notify all the components by the end of the process.

## III. Details about the prototype

Our prototype is based on Redis and developed in Javasrcipt. It is organized into the following pseudo classes:

- ServerClass: there are three cases for servers; a main server for the Redis protocol itself through the publication-subscription interface, a data server for storing the input/output data, and a code server to retrieve the necessary code for the execution of an application.
- DataManager: it defines an instance of the class ServerClass. It also defines the methods for loading files on/from the Redis servers, for the execution of a code (binaries or scripts).
- MachineClass: it defines the properties of a machine (operating system, available memory, processor type,...)
- BrokerClass, CoordinatorClass, WorkerClass, MonitorClass and CheckerClass define respectively the behavior of *the broker, the coordinator, worker, monitor and checker*.

The application that serves to validate our prototype is the well known MapReduce word-count. The application with the dependencies between tasks is depicted by an acyclic graph. Each node in the graph represents a task and each edge between two nodes represents a dependency between two tasks. A node is described in terms of inputs, outputs and code to execute. The input text file is split in 4 pieces according to a Shell script, then four mappers count for the number of times each word appears, then two reducers merge the partial result one time, and at last a final reducer step produces the final result. Both the reducer and the mapper are Python codes. These codes, as well as the Shell script are downloaded, on the fly by the workers (reducers and mappers) from the Redis code server. The workers also download the input files from the Redis data server and produce output files that are also put into the Redis data server. Note that the interaction protocol is written in Javascript that call external scripts in Python and Shell.

The javascipt libraries used by our prototype are:

- jsnetworkx: JSNetworkX is a port of the popular Python graph library NetworkX. It's a software package for the creation, manipulation, and study of the structure, dynamics, and function of complex networks. With NetworkX you can load and store networks in standard and nonstandard data formats, generate many types of random and classic networks, analyze network structure, build network models, design new network algorithms, draw networks. In our case, we use the package to depict our application graph as a di-graph and we also use the methods for retrieving the source/sink of a node for instance.
- redis and redis-cli (for the tests of the databases of the

Redis servers). This is the javascrip modules that implement the javascript interface with the Redis server. The libraries encapsulate the Redis commands to interrogate the Redis databases;
- fs and exec: these modules provide an interface for file handling and process handling. For instance, we need to fork a new process (by an equivalent of an unix exec commmand) to start the computation that a worker have to accomplish (Shell or Python scripts)
- zlib: this module is used to compress the files before we send them to the redis servers in order to save room;

The simulation of the Javascipt word count application that run under our framework requires Nodejs. Node.js is an open-source, cross-platform runtime environment for developing server-side web applications. Node.js applications are written in JavaScript and can be run within the Node.js runtime on OS X, Microsoft Windows, Linux, FreeBSD... That means that our current prototype is not self-content, meaning that the javascript do not fully run in the browser, interpreted by the browser. However, the prototype is written in standard Javascript and we made this choice because it is easily, from our point of view, to debug under Nodejs rather than under the browser. We are also waiting that all the required module for our prototype will be available as full Javascript codes and not pre-compiled librairies for Nodejs. It is just a question of time.

The most interesting part of Node.js is that it provides an event-driven architecture and a non-blocking I/O API designed to optimize an application's throughput and scalability for real-time web applications. It uses Google V8 JavaScript engine to execute code, and a large percentage of the basic modules are written in JavaScript but not all the ones required by our prototype. The event-driven architecture of Nodejs calls for round trips with the formal modeling. In fact the modeling with Petri nets may takes into account such consideration. With a Petri net you define actions that take place simultaneously and correspond to tokens that are fired at the same time. Guards can also be associated to transitions to simulate the event-driven nature of a problem. However, the graphical nature of Petri nets does not translate immadiately to a Javascript code and a programmer has to check many points that require good skill in programming and also in Petri nets modeling. At least, the non-blocking nature of javascript function lead to implementations where the differents calls of a function need to be encapsulated into a callback, that is not really obvious for the development and tricky.

## IV. Related works

In this section we mainly summarize works related to data management with a focus on desktop grid data management systems. Grid computing researchers have developed data management systems such as Stork [5], SRM [6], JuxMem [7], Freeloader [8]. These systems have demonstrated their robustness and reliability in grid environments. In addition, Bitdew [9] and GatorShare [10] provide a programmable framework for data management in desktop grids. In this paper we argue that these systems should be part of a more general interaction scheme that remains to be built because we are focussing on scheduling, monitoring, certifying the results. The data management is through Redis servers than deal with the Pub-Sub paradigm for the first one, the code app store for the second one and the input/output data store pour the third one.

BitDew [9]

Active Data []

*** Others work related to the two previous one come here. ***

In [2] authors focussed on the BonjourGrid [11], [12], [13] meta desktop-grid middleware and demonstrated how a user can select on demand, his favorite computing middleware (BOINC, Condor, XtremWeb) as well as his favorite data manager (Stork, Bitdew) under the supervision of an interaction scheme implemented on top of Redis. In this paper we start with a new interaction scheme, RedisDG, introduced in [14] which is also implemented on top of Redis but it revisits the interactions in order to facilitate the development in Python as well as in Javascript, two popular languages in the Web ecosystem.

According to Fedak et al. in chapter 11 of [15], in Desktop Grid environments, basic data-management tasks such as reliably storing large data-sets are very difficult to accomplish, first because of the volatility of nodes. Second, data privacy and security must be enforced on Desktop Grids because we deal with untrusted computers. The data protection mechanism may add non-trivial overhead when processing large volumes of data. Third, since the resources are geographically distributed, the design of a scalable data-intensive solution on these systems is an issue. In this work, we also focus on the latter issue, but at a MODESTE scale since we have built a prototype and not a production system. The other ones are left as part of either the computing or data exchange sub-systems.

## V. Conclusion and future work

In this article we have explained our motivations and initial work for doing Data Science in using browsers as data and computing nodes. We assume that a large collection of devices (smartphones, tablets, connected watches...) can be 'federated' as we have done in the past with PCs to form the desktop grid paradigm. We also assume that the federation will happen because all these devices can run a web browser such as Google Chrome or Firefox and because these browsers implement all the functionalities, inherited from the Web ecosystem, we found on our desktop machines. In some ways we are renovating the desktop grid paradigm where volonteer nodes participate into computation. We proposed a methodology that considers first a high level of abstraction for the interactions between components that have been identified over time by colleagues working in the cluster and grid fields, all sub-fields included. Second we implemented a prototype that serve to measure the technical difficulties in implementing the abstration and it reveals the round-trips required to enrich the model.

Our future work will mainly consider alternatives to the management of data. In the current prototype we are using

Redis servers to store codes, input and output data, but without direct communication between browsers. This situation is like what the community has done at the beginning in desktop grid computing but it is not always the best choice for performance matters. We will refine our formal modeling in order to take into account direct communication between browsers explicitely. The BitDew system [9] will serve as a use case. Moreover we are also guessing that the Active Data model [] to take into account the data life cycle could be unified with our interaction scheme. The two systems rely on Petri Nets and the difficulty is on how to define a coupling or composition of the two systems. The general objective is to progressively add or to glue sub-components for data management into the interaction scheme that depicts how core components (compute on data, certify results, monitoring, scheduling tasks) in order to offer the same services as we can find them nowadays on clusters, grids and clouds.

## Acknowledgment

This work has been done partly under the Wendelin grant from the ministry of industry in France (Programme Investissement d'Avenir). This work has also been conducted with the support of the President's International Fellowship Initiavite (PIFI) of the Chinese Academy of Science that founded months of invited professor with CSTNET for two authors.

## References

[1] D. Cassou, S. Ducasse, and N. Petton, "Safejs: Hermetic sandboxing for javascript," *CoRR*, vol. abs/1309.3914, 2013. [Online]. Available: http://arxiv.org/abs/1309.3914

[2] W. Saad, L. Abidi, H. Abbes, C. Cérin, and M. Jemni, "Wide area bonjourgrid as a data desktop grid: Modeling and implementation on top of redis," in *26th IEEE International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2014, Paris, France, October 22-24, 2014*. IEEE Computer Society, 2014, pp. 286–293. [Online]. Available: http://dx.doi.org/10.1109/SBAC-PAD.2014.50

[3] P. T. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, 2003.

[4] H. Abbes and J.-C. Dubacq, "Analysis of peer-to-peer protocols performance for establishing a decentralized desktop grid middleware," in *Euro-Par Workshops*, ser. Lecture Notes in Computer Science, E. César, M. Alexander, A. Streit, J. L. Träff, C. Cérin, A. Knüpfer, D. Kranzlmüller, and S. Jha, Eds., vol. 5415. Springer, 2008, pp. 235–246.

[5] T. Kosar and M. Livny, "Stork: Making Data Placement a First Class Citizen in the Grid," *Distributed Computing Systems, International Conference on*, vol. 0, pp. 342–349, 2004.

[6] A. Shoshani, A. Sim, and J. Gu, "Storage Resource Managers: Middleware Components for Grid Storage," *In Nineteenth IEEE Symposium on Mass Storage Systems*, 2002. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.19.3515

[7] G. Antoniu, L. Bougé, and M. Jan, "JuxMem: An Adaptive Supportive Platform for Data Sharing on the Grid," *Scalable Computing: Practice and Experience*, vol. 6, no. 33, pp. 45–55, Nov. 2005. [Online]. Available: http://hal.inria.fr/inria-00000984/en

[8] S. S. Vazhkudai, X. Ma, V. W. Freeh, J. W. Strickland, and et al., "Freeloader: Scavenging desktop storage resources for scientific data," *In proceeding of supercomputing*, 2005.

[9] G. Fedak, H. He, and F. Cappello, "BitDew: A data management and distribution service with multi-protocol file transfer and metadata abstraction," *Journal of Network and Computer Applications*, vol. 32, no. 5, pp. 961–975, Sep. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.jnca.2009.04.002

[10] J. Xu and R. J. O. Figueiredo, "Gatorshare: a file system framework for high-throughput data management," *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, HPDC 2010, Chicago, Illinois,USA, June 21-25, 2010*, pp. 776–786, 2010.

[11] H. Abbes, C. Cerin, and M. Jemni, "Bonjourgrid: Orchestration of multi-instances of grid middlewares on institutional desktop grids," *Parallel and Distributed Processing Symposium, International*, vol. 0, pp. 1–8, 2009.

[12] H. Abbes, C. Cérin, and M. Jemni, "Bonjourgrid as a decentralised job scheduler," in *APSCC*. IEEE, 2008, pp. 89–94.

[13] H. Abbes, C. Cérin, M. Jemni, and W. Saad, "Toward a meta-grid middleware." *Journal of Internet Technology, Volume 11 No1*, 2010.

[14] L. Abidi, C. Cérin, and M. Jemni, "Desktop grid computing at the age of the web," in *Grid and Pervasive Computing - 8th International Conference, GPC 2013 and Colocated Workshops, Seoul, Korea, May 9-11, 2013. Proceedings*, ser. Lecture Notes in Computer Science, J. J. Park, H. R. Arabnia, C. Kim, W. Shi, and J. Gil, Eds., vol. 7861. Springer, 2013, pp. 253–261. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-38027-3_27

[15] C. Cerin and G. Fedak, *Desktop Grid Computing*, 1st ed. Chapman and Hall-CRC, 2012.